NAIC-ID(RS)T-0024-95

NATIONAL AIR INTELLIGENCE CENTER

DESIGN AND IMPLEMENTATION TECHNIQUES OF THE
8086 C DECOMPILING SYSTEM

by

Chen Fuan, Liu Zongtian, Li Li

**Approved for public release:
distribution unlimited**

HUMAN TRANSLATION

NAIC-ID(RS)T-0024-95 25 May 1995

MICROFICHE NR: 95C000330

DESIGN AND IMPLEMENTATION TECHNIQUES OF THE 8086 C DECOMPILING SYSTEM

By: Chen Fuan, Liu Zongtian, Li Li

English pages: 24

Source: Xiaoxing Weixing Jisuanji, Vol. 14, Nr. 4, 1993

Country of origin: China
Translated by: SCITRAN
F33657-84-D-0165
Requester: NAIC/TATA/Keith D. Anthony
Approved for public release: distribution unlimited.

GRAPHICS DISCLAIMER

All figures, graphics, tables, equations, etc. merged into this translation were extracted from the best quality copy available.

DESIGN AND IMPLEMENTATION TECHNIQUES OF THE 8086 C DECOMPILING SYSTEM

/10*

Chen Fuan Liu Zongtian Li Li

ABSTRACT

This paper presents the structural design of 8086 C decompiling systems and the no symbolic information decompiling techniques of C language which have been implemented, that is, library function pattern recognition techniques, Sub C intermediate language, symbolic execution techniques, rule based data type recovery techniques, as well as rule based ABC program transformation techniques, and so on. The authors have applied the techniques described above on PC type micro machines to implement 8086 C language decompiling systems. The system in question is capable of taking Microsoft C (Ver 5.0) small memory type no symbolic information 8086 object code programs and translating them into C language programs with equivalent functions.

KEY WORDS: Decompiling system Symbolic execution Pattern recognition of library functions Data type recovery Program transformation

I. INTRODUCTION

Decompiling system functions are to take low level language code programs and translate them into high level language programs of equivalent function, making programs easy to read, understand, safeguard, and revise. Following along with the development of software engineering, analysis and understanding of existing /11 software as well as its revision and reuse have already received a full measure of attention from people. In particular, software development levels in China lag behind developed countries. Most system software is based on introductions from abroad. The dissection and analysis of these software and their

---

* Numbers in margins indicate foreign pagination. Commas in numbers indicate decimals.

signification and transplantation, then, seem very important. Because of this, people have compelling need for the tools of analysis and understanding of this type of software such as decompiling systems.

Research with regard to decompiling techniques in China and abroad already has a history of more than twenty years [1,2,3,4]. However, up to the present day, there is still no practically useful decompiling system which has come out. The primary cause for this lies in the relatively great difficulty of researching no symbolic information object program decompiling techniques. The authors, on Dual--68000 micro machines, developed 68000 C decompiling systems [5]. The systems in question are capable of taking 68000 object code programs formulated by C compilations (including such symbolic information as overall variable names and library function names) and translating them into C language programs. However, as far as the majority of software in actual use is concerned, the object code programs do not include any symbolic information. In order to raise the suitability of decompiling systems, and, in conjunction with that, make them practically useful, in a 75 period, we carried out research aimed at C language decompiling techniques associated with no symbolic information. In conjunction with this, on PC type micro machines, implementation was made of an 8086 C language decompiling system. The system in question is capable of taking no symbolic information 8086 object code programs in Microsoft C (Ver 5.0) small memory forms and translating them into C language programs with equivalent functions.

II. FUNCTION STRUCTURE OF 8086 C LANGUAGE DECOMPILING SYSTEMS

In decompiling of no symbolic information object programs, primarily, two large problems are resolved. One is recovery of such symbolic information as variable and library function names as well as data types. The second is restoration of program control

structures. Moreover, this must be carried out on the foundation of symbolic information recovery. Therefore, 8086 C language decompiling system structure design is composed of the five functional modules of decompilation and library function recognition, symbolic execution, data type recovery, AB program transformation, and C program transformation. The system function structure schematic is as shown in Fig.1. In it:

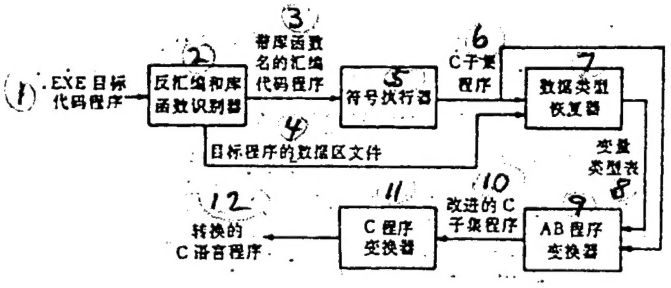


Fig.1 8086 C Decompilation System Function Structure Line and Block Chart

Key: (1) E E Object Code Program (2) Decompilation and Library Function Recognition Device (3) Compilation Code Carrying the Name of Library Function (4) Object Program Data Area Document (5) Symbolic Execution Device (6) C Subset Program (7) Data Type Recovery Device (8) Variable Type Table (9) AB Program Transformation Device (10) Improved C Subset Program (11) C Program Transformation Device (12) Translated C Language Program

1. Library Function Recognition Module

Decompilation is carried out on E E type object code programs. At the same time, C library function recognition is carried out. In conjunction with this, it is possible to distinguish user defined

functions, outputing compilation code programs which carry library function names (containing user defined function names). At the same time, data area documents are formulated in object programs.

2. Symbolic Execution Interpretation Module

Compilation code programs output from the library function recognition module are taken and translated into intermediate language programs. The intermediate language in question is a C language subset containing only two types of control statements--goto and conditional goto.

3. Data Type Recovery Module

Input is data area documents associated with C subset programs and object programs. Going through a collecting and synthesis of variable type information dispersed in programs, the data types associated with various variables are determined. In conjunction with this, variable type table forms are taken and output.

4. AB Program Transformation Module /12

On the basis of variable type tables, address expression forms in C subset programs are taken and translated into C language expression forms associated with number set elements or structural elements. In conjunction with this, type declaration statements associated with external variables and local variables are produced.

5. C Program Transformation Module

C subset program control structures are taken and transformed into such control statements as if-else, while-do, do-while, and so on, formulating C language programs possessing good structures.

Operations associated with the modules described above all produce intermediate text in magnetic disc document form. Users are capable of separately utilizing intermediate text and final C language programs to carry out software dissection and analysis

III. LIBRARY FUNCTION PATTERN RECOGNITION TECHNIQUES

During decompilation of no symbolic information object code programs, the symbolic information recovery plan designed is: first of all, in decompilation processes associated with object code programs, apply pattern recognition principles [6] to carry out C library function recognition, thereby recovering library function symbolic names used in programs. Formulate compilation code programs carrying library function names. The second step is--on the basis of C subset intermediate language programs formulated in symbolic execution--to carry out variable and data type recovery. In 8086 C language decompilation systems, the authors analyzed C library function composition forms in object code programs. Designed library function pattern recognition techniques include the analysis and restoration of C library function recognition characteristics, design of pattern matching methods, setting up C library function characteristic code recognition tables, as well as program implementation.

1. Analysis and Recovery of C Library Function Recognition Characteristics

The authors took command operating code sequences to act as C library function recognition characteristics. This is because:

(1) No matter what type of composition form C library function code is or what type of utilization conditions it is under, the command operating code will not change. Only certain command operation numbers can give rise to variations;

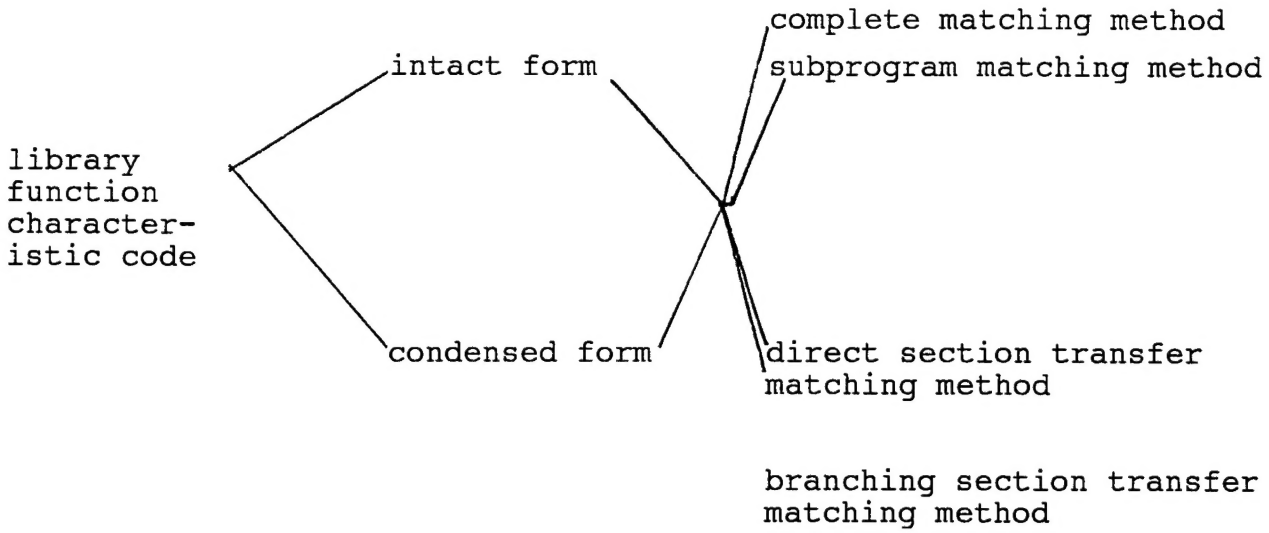
(2) Different library functions--due to their different functions--have different command operating code sequences associated with their composition;

(3) On the basis of 8086 compilation command coding rules, in command operating codes, there are contained codings associated with operation number types and addressing forms. In this way, the operation number codes are removed from certain command codes. However, the full characteristics of operation numbers have certainly not been lost.

Recovery methods associated with C library function recognition characteristics are: under the principles of characteristic adequacy, primary and supplementary characteristics associated with library function code are both picked up, that is, both primary characteristic codes are recovered from primary subprogram text and supplementary characteristic codes are recovered from transferred function subprograms (if they exist). The authors utilized this type of method to set up characteristic code tables associated with recognized C library functions. On the basis of whether or not library function codes were completely recovered, characteristic codes can be divided into complete--taking complete codes to act as characteristic codes, as well as condensed codes--recovering partial codes to form characteristic codes under the principles of characteristic adequacy.

2. Pattern Matching Methods Associated with C Library Function Recognition

On the basis of C library function code composition forms and characteristic code recovery methods, the authors opted for the use of static matching and dynamic matching methods, analyzing and designing eight types of pattern matching methods associated with C library function recognition:



For example, subprogram matching methods: after the carrying out of subprogram transfer code searches on library function characteristic codes and library function codes waiting to be recognized, subprogram transfer commands in dynamic execution programs were gone through in order to determine the subprogram entry addresses. After this, the entry addresses in question were compressed into stacks. At the same time, supplementary /13 characteristic pattern matching addresses (preset in tables) were also taken and compressed into stacks. After pattern matching to the same main characteristic, pattern matching of supplementary characteristics was also carried out. For example, with regard to recognition of library functions having both main subprograms and supplementary subprograms, the main subprograms generally opt for the use of complete matching methods for handling. However, supplementary characteristic codes, by contrast, opt for the use of subprogram matching methods in order to process them. On the basis of the pattern matching methods discussed above, the authors set up pattern matching method tables associated with the recognition of various individual C library function characteristic codes. The majority of pattern matching methods associated with function recognition are formed from syntheses of multiple types of matching methods.

3. C Library Function Characteristic Code Recognition Tables and Implementation Methods.

In order to implement the design ideas associated with C library function pattern recognition, in 8086 C decompiler systems, the authors set up a C library function characteristic code recognition table (see as shown in Fig.2). It is nothing else than an aggregation of library function matching patterns which can be recognized by the system. The table in question is associated with matching patterns in which each element stands for a library function. Library function matching patterns are primarily composed of two parts--library function characteristic code tables and pattern matching form tables. Here, characteristic code tables are composed of recovered characteristic code sequences. Moreover, pattern matching code tables are composed of library function pattern matching operation coding. It includes such information as pattern matching form coding, matching start addresses in characteristic code tables, code matching lengths, as well as determination forms associated with code matching addresses in object code programs.

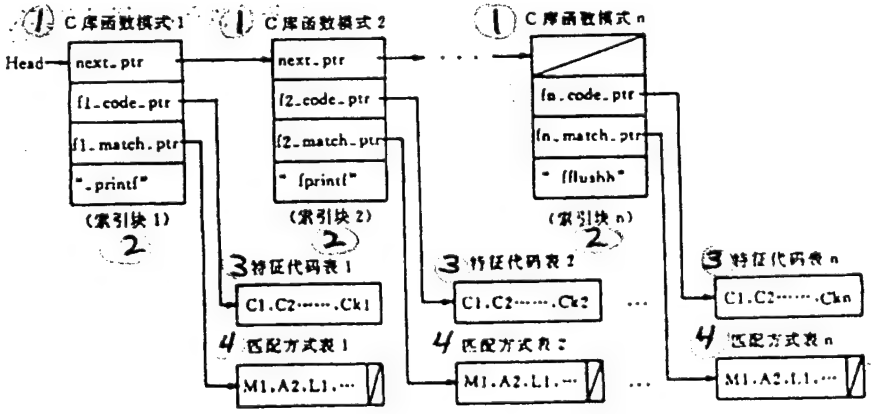


Fig.2 C Library Function Characteristic Code Recognition Table Structural Schematic (key on following page)

Key: (1) C Library Function Pattern (2) Index Module (3) Characteristic Code Table (4) Matching Form Table

Implementation methods associated with C library function pattern recognition techniques are: with regard to program code decompilation processes, simultaneously set up transfer entry address tables associated with subprograms utilized in programs.

After that, take start codes corresponding to various entry addresses and characteristic code recognition table elements and carry out function matching comparisons--respectively, taking recognized library function symbolic names and adding them in to entry address tables. Ones which cannot be recognized, by contrast, are placed in as yet unrecognized character strings. With regard to subprograms which have not yet been recognized, it is possible, on the basis of address range areas, to separate out user defined functions. Finally, take recognized result character strings and replace subprogram transfer entry addresses, formulating compilation code programs carrying C library function names (including user defined function names). For a detailed introduction to related library function pattern recognition techniques, see [7].

IV. SYMBOLIC E ECUTION TECHNIQUES ASSOCIATED WITH INTERMEDIATE LANGUAGE FORMULATION

The introduction of intermediate languages into decompiling is a help in lowering the complexity of decompiling processes, rational division of the tasks associated with various stages of decompilation, as well as increasing the transplantability of decompilation systems. In decompiling systems, intermediate languages are geared toward high level language translation. They are situated between low level languages and high level languages. They are not only more advanced in scope than low level languages. They must also include basic structures of high level language. The form of expression and structure must be appropriate to program

analysis, restructuring, and translation. In 8086 C decompiler systems, the authors defined Sub-C intermediate language. It is situated between compilation code and high level language. It only includes two types of control statements associated with C grammar forms--goto and conditional goto. The valuation statement forms are the same as C language. However, expression forms for variables not only permit C language forms but also address expression forms. Address expression forms are a type of abstract form situated between calculation variable address code sequences in object code and variable applications in high level language. Generally, they are formed by using a "+" sign to connect subexpression forms associated with variable base addresses and calculation offset addresses. For example, the address expression form ae(A[V1][V2] associated with array elements A[V1][V2] has an intermediate language form of: /14

$$[ae(V1)*N1+ae(V2)*N2+addr(A)]-L$$

In this, ae (Vi) expresses the address expression for Vi. add (A) is an array header address. Ni is invariable. L is array element length.

In 8086 C decompiler systems, the authors opted for the use of symbolic execution techniques taking compilation code and translating it into intermediate language programs. Symbolic execution is initially applied to program empirical verification and debugging [8]. The basic idea is: simulate program execution processes, introduce symbolic objects to act as input, replace the actual numerical value objects, and make programs run in a symbolic environment formulating a program using input symbolic representations. The techniques in question have been applied in 68000 C decompilation systems [9]. As far as the carrying out of symbolic execution on computer programs is concerned, it is necessary to specify the symbolic execution environment as well as the symbolic execution meanings associated with program statements.

1. 8086 Compilation Code Symbolic Execution Environment Definitions Are:

three address registers: r0, r1, r2;

two operation number attribute registers: outop1, outop2 one two character long character string register T;

one character string stack with stack top indicator psp;

certain single character string temporary units: sax, sal, sah, sbx, scx, scl, sdx, sdl, store [0-2];

one output document with indicator out always pointing toward the header address of the area which can be written to in front of the document in question.

2. Attribute Definition

In order to interpret and execute 8086 compilation commands in a symbolic execution environment, it is necessary to give attribute definitions associated with compilation command operation codes, relational codes, and operation numbers. For example, in the case of operation codes ADD, ADC, FADD, and FIADD, their attribute properties are "+". However, the attribute properties of SHL and SAL are displacement left "<<". Again, for example, attribute properties of relation codes JA and JNBE are ">", and so on, and so on.

3. Description of 8086 Compilation Command Symbolic Execution Semantics

Scanning compilation code text, and, in conjunction, executing in accordance with compilation command semantics in symbolic environments, it is then possible to formulate intermediate language programs equivalent to compilation code meanings. For example, as far as compilation commands add v1, dx

are concerned, in them, v1 is a variable name, dx is a register, and the order of symbolic execution is:
(1) adopt the add attribute property "+"; (2) take the character string "v1" and grant symbolic variable sv1; (3) use "+" to connect the contents between sdx and sv1, and the results exist in sdx.

Assume that sdx was originally stored "xyz". Then, after symbolic execution of the commands in question, "xyz+v1" is stored in sdx. If the follow on command is: sub dx, v2; then, after executing this command, the character string "v2=v2- (xyz+v1)" is output to the output document out.

4. Symbolic Execution Function Sequence

Symbolic execution is not carried out using 8086 compilation program command sequence; however, it is in accordance with program control sequence going through each possible execution path. To this end, it first divides up basic command blocks. On each basic block, relational analysis is carried out in association with fixed values applied to common registers in order to determine execution sequences associated with the various blocks. In conjunction with this, data flow analysis is carried out in order to determine whether or not basic blocks will accept contents or pass them on. With regard to code in basic blocks, it is run in accordance with symbolic environment semantic interpretations. Output symbolic strings are nothing else than corresponding Sub C programs. Compilation command sequences without inputs and outputs are called basic blocks. Execution functions between various basic module blocks can be carried out in accordance with program flow chart depth prioritized sequences.

Symbolic execution is carried out in accordance with interpretation forms. Each command is scanned in order in each basic block. Each is treated as a single character, that is, it is run in accordance with corresponding semantic actions. The results

of execution are that, in the output document out, one obtains Sub C intermediate language programs. Besides that, on the foundation of the symbolic execution of 8086 compilation language, various types of variable address information are retained. Addressing methods associated with certain variables are maintained unchanged in order to facilitate the supplying, for data type recovery, of variable data type application information which is as complete as possible.

/15

V. RULE BASED DATA TYPE RECOVERY TECHNIQUES

After variable type explanation statements associated with high level language programs go through compilation, no object code is produced. Compilation devices simply distribute variables to memory storage spaces on the basis of information supplied by explanation statements, set up symbolic tables, and, at the same time, take queries about variables in programs and translate them into corresponding memory storage unit queries or calculations associated with address expressions. Moreover, after compilation, symbolic tables generally do not remain in object code. In this way, data type recovery is only able to apply information dependent on such things as variable memory storage unit distribution forms and query forms. In 8086C decompiler systems, the authors studied data type recovery techniques based on rules, that is, through scanning readings of Sub C intermediate language programs, data area documents associated with object programs are synthesized, and such application information as distribution forms for variable memory storage units, query forms, and so on, are gathered and analyzed as well as the mutual relationships between them. After that, comprehensive type analysis is carried out, thereby recovering variable data types. The authors already summarized collection and analysis rules for application information associated with 17 variable data types as well as comprehensive analysis rules for 8 data types (for details see [10]). This

section will explain the utilization of variable memory storage unit distribution form information as well as citing examples to explain data type recovery techniques based on rules.

1. Utilization of Variable Memory Storage Unit Distribution Form Information

In Microsoft C (ver 5.0) small memory patterns, object program code area and data area lengths are ≤ 64K bytes. Data area variable memory unit distribution forms are as shown in Fig.3. It is possible, on the basis of variable distribution forms in data areas, to divide them into constant variables, overall variables, and local variables. Therefore, storage locations for overall variables during program operation are fixed. However, storage units associated with local variables and form parameters are distributed in stack space. This type of storage distribution follows along with functional transfer dynamic variations. However, it requires knowing beforehand total amounts of local variable memory.

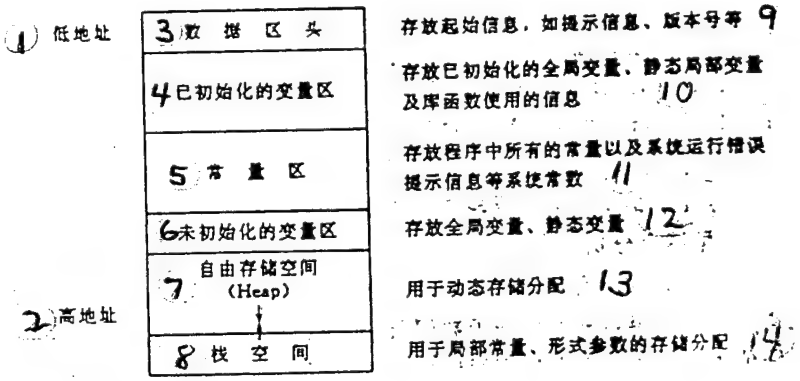


Fig.3 Data Area Variable Memory Unit Distribution Form Schematic

Key: (1) Low Address (2) High Address (3) Data Area Header (4) Already Initialized Variable Area (5) Constant Variable Area (6) non-initialized Variable Area (7) Free Memory Space (HEAP) (8) Stack Space (9) Initial Storage Information Such As Announcement Information, Edition No., And So On (10) Stores Information Associated with Already Initialized Overall Variables, Static Local Variables, as well as Library Function Utilization (11) System Constants Such as Constant Variables in Stored Programs as well as System Operation Error Announcement Information, and so on (12) Stores Overall Variables and Static Variables (13) Used in Dynamic Memory Distribution (14) Used in

Memory Distribution Associated with Local Constant Variables and Form Parameters

2. Collection and Analysis of Variable Data Type Application Information

Variable data type application information is principally reflected in:

(1) Variable unit storage and retrieval length (that is, byte number);

(2) Variable unit operation character characteristics;

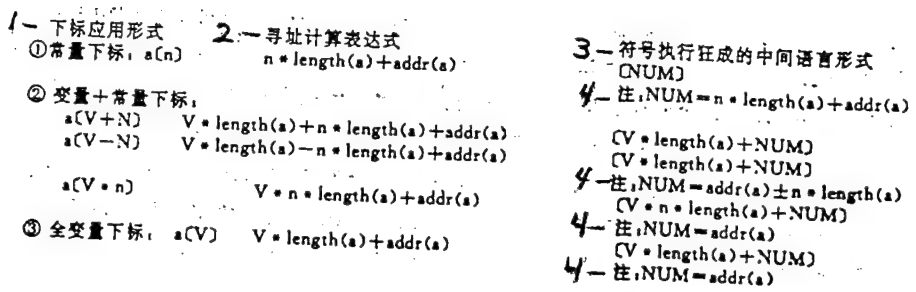
(3) Variable unit addressing form (Sub C addressing expression forms);

(4) Value transmission and citation relationships between variables;

(5) Transmission relationships between function form parameters and actual parameters.

Going through individual word analysis of variables, it is, then, possible to gather memory unit operation character characteristics and unit lengths for them.

For example, as far as the gathering of array variable application information is concerned, it is primarily the /16 collecting of array dimension numbers and element lengths. The reason is that arrays, in reality, express linear memory storage forms associated with certain variable types. In C language, with regard to array application realizations--in queries against array elements, this type of query is implemented through the utilization of array subscripts. On the basis of their compilation code calculation forms, subscripts can be divided into constant subscripts, constant + variable subscripts, and full variable subscripts. Assume that there is a one dimensional array a [m]. add (a) is the array memory start address. Moreover, length (a) expresses the type length associated with element a (byte number). In that case, the subscript application status of it is:



Key: 1-Subscript Application Form 2-Addressing Calculation Expression Form 3-Intermediate Language Forms Created by Symbolic Execution (1) Constant Subscript (2) Variable + Constant Subscript (3) Full Variable Subscript 4-Note:

Due to compilation optimization, constant operations which exist in array subscript addressing are all directly calculated results when compiling. As a result, in compilation code, as long as one opts for the use of full variable subscripts for array queries, it is only then possible to retain complete calculation processes. Thus, whether or not array variable information collection is correct depends on whether or not option is made in original programs for the use of full variable subscript queries of array elements. If option is made for the use of constant subscript queries of single array elements, then, recovery is easy as simple variables. Rules for collection and analysis of array variable application information are:

R1 Assume add is a single variable address. With regard to VAR single word addressing expression forms in symbolic execution code, they are [expression form + add]. If the expression form contains IN * n terms, then, it is possible to determine that add is an array variable address. In it, IN is the word VAR or the word FUN. n is a full form constant.

The R1 rule is not capable of determining one dimensional character arrays. Because the character variable length is 1, a[V + add] addressing expression form is [V + add (a)]. No V * n terms exist. Therefore, one has the rule:

R2 Assume add is a variable unit address. The addressing expression form reading in the single word VAR is [V + add] tL. If V is not an indicator of variable and L=1, then, add is the header address of a character array.

As far as collection of variable application information from mutual value transmission or citation relationships between variables is concerned, it is then necessary to go through scanning reads of intermediate language statements, and, if they are not single words only then is it possible to collect them. With regard to statements adequate to reflect relationships between variables, there are function transfer and valuation statements.

3. Variable Data Type Comprehensive Analysis

After gathering of application information is completed, relevant information is all stored in variable unit application information tables. In these, there are full variable record chains in accordance with address sequence arrangements, all user function record tables, and, in each function table, are contained form parameter tables, local variable record chains, function reentry values, and so on. However, due to the fact that, in variable application information gathering stages, application information gathered on various individual variables is usually independent and unilateral--for example, composite structure variables. Relationships between variable units certainly have not had analyses carried out on them. Because of this, it is necessary to carry out comprehensive type analysis on recorded variable chains which one already has. At the same time, one must carry out a merging with corresponding variable units. Only then is one able

to recover accurate variable types. Using comprehensive analysis of indicator application information as an example:

Indicator variable application information, in actuality, reflects application information associated with variables which have been directed. However, due to the fact that information collection processes are carried out in accordance with code sequences, it is not possible to specify variables which are directed during indicator applications. Because of this, after completing information gathering, one should take indicator application information and transmit it into application information tables associated with each individual variable unit and directed variable chains

With regard to function indicator type form parameters, one ought also to carry out indicator application information transmission processing. The reason for is that this type of function transfer opts for the use of real parameters which, in actuality, are variable addresses. Moreover, application information associated with form parameters reflects--in a transfer environment--application information associated with the variables in question. In this way, variable information in a transfer environment will not then be isolated in the interior of transferred functions.

After carrying out application information transmission processing on all indicator variables or indicator type form parameters, there is still a possibility of having indicator variables which have not yet been transmitted. This is because, when variables which indicator variables themselves point toward are also indicator types, after going through transmission processing, there is the possibility of producing new indicator variables associated with application information which has not been transmitted. Because of this, it is necessary to repeat /17 transmission processes associated with indicator application

information, right on until there are no indicator variables which have not been transmitted and stopping.

The accuracy of variable data type recovery depends on whether or not variable application information gathering is complete. At present, there still exist inaccuracy problems associated with variable data type recovery. Imperfect variable memory storage unit application information is the primary factor creating inaccuracy. Moreover, variable application information imperfections are primarily due to incomplete variable applications in original programs and compilation code optimization, as a result, producing data type recovery isolog phenomena.

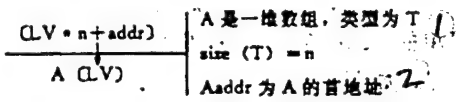
VI. ABC PROGRAM TRANSFORMATION TECHNIQUES

In 8086C decompiler systems, the authors utilized rule based program transformation techniques in order to carry out carry out program statement translation.

1. A Transformation and B Transformation [11] [12]

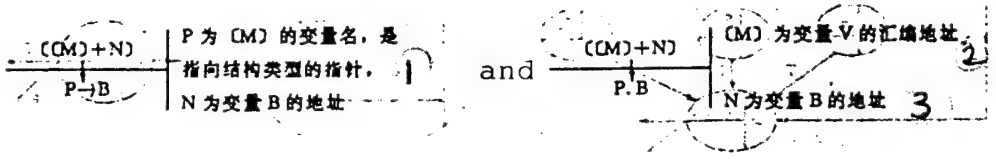
In C compiler programs, utilizations of variables, array elements, and structural components all are translated into calculation memory storage address command strings. Going through symbolic execution, Sub C programs are formulated, merely taking these calculation address command strings and translating them into a number of address expression forms. The authors did specialized processing on these calculation address expression forms, utilizing methods associated with program form transformations and taking them and transforming them into corresponding variable names, array elements or C language statement expression forms of structural component queries. This type of transformation is called A transformation.

Transformation rules designed by the authors are divided into three sections. The top section is changed forms. The bottom section is formulated forms. The right side is transformation conditions. For example:



Key: (1) A is a one dimensional array. Type is T. (2) Aaddr is header address of A

Address expression form types are very numerous. Address expression forms of the same type--because transformation conditions are not the same--have transformation structure forms which are also different. As far as these same address expression forms are concerned, there is also the possibility of transforming into structural forms which are the same. For example:



(1) P is the variable name of [M] which refers to indicator structure type. N is the address of variable B. (2) [M] is the compilation address of variable V. N is the address of variable B.

With regard to translations of address expression forms not containing sub expression forms:

Assume: [V1 * n1+... +Vk * nk+add]

Look up addresses ≤ add variable type tables. For example, find the address of variable gv to be add'. add' ≤ add. gv is a k dimension structural array. Dimension numbers, respectively, are

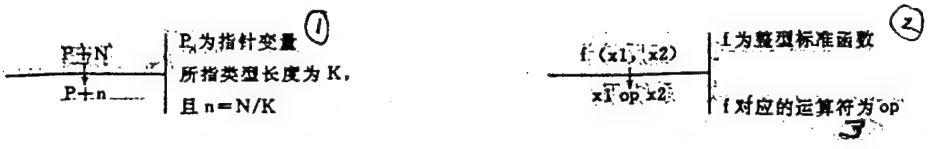
d1, d2, ... , dk. Then, the forms described above can be translated as:

gv [V1 * m1] [V2 * m2]... [Vk * mk]. ej

In this, the corresponding amount of offset associated with m1=n1/d1, m2=n2/d2,... k mk=nk/dk, ej is add--add'. If mj=1, then [Vj * Mj] is simply designated [Vj].

In 8086C decompiler systems, there are 10 address expression form transformation rules utilizing unified algorithms in order to complete transformations associated with different transformation rules for various types of address expression forms.

In Sub C programs, after going through A transformation, there still remain a number of non C statement expression forms. It is necessary to go through further revisions. The authors added B transformation in order to take non C statement expression forms and translate them into C forms. For example, B transformation rules:



Key: (1) P is an indicator variable. Indicated type length is K, and n=N/K. (2) f is complete form standard function. Operational character corresponding to f is op.

/18

2. C Transformation

C transformation is a program structuring transformation. It takes Sub C program nonstructured control forms and transforms them into expressions by such statements as if-else, for, while- do, do-while, possessing C language programs with good structures. The

transformation techniques in question have already been introduced in [13]. Here, only transformation methods associated with 8086C decompiler systems are introduced. Using Sub C flow charts, switch structural forms expressed by SCG are as shown in Fig.4:

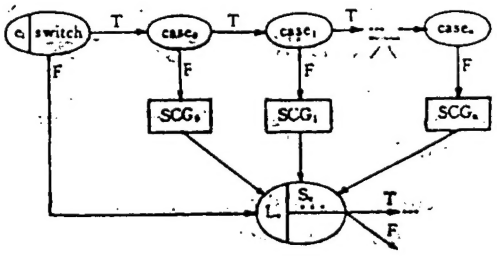


Fig.4

Taking Sub C intermediate language and translating it into SCG switch structures, the difficulty lies in how to determine common follow on nodes Le and SCGi non-normal exits as well as non-normal entries under the various branching cases associated with SCGi. Opting for the use of basic block break downs, break statements or goto statements replace these non-normal exits and entries, making each individual SCGi turn into an SCGi' which only has one entry and one exit. After going through C transformation, each SCG is a simple SCG nodal point. Then, standard switch statement SCG are as shown in Fig.5. Following that, using switching rules, they are simplified to a single node as shown in Fig.6:

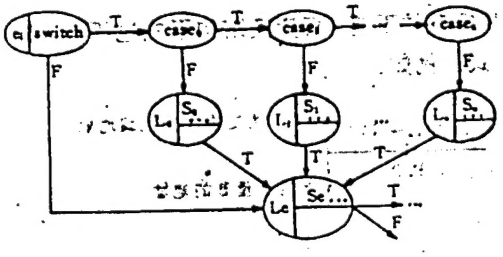


Fig.5

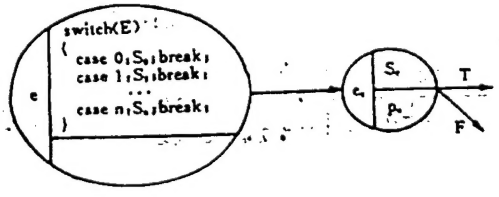


Fig.6

VII. CONCLUDING REMARKS

The characteristics of 8086C language decompiler system implementation are:

(1) Exploring ways to resolve symbolic information recovery problems: in the recovery of object programs applying library function pattern recognition techniques, utilizing library function names, compilation code programs are formulated carrying library function symbolic names. As far as application data types are concerned, one should use information gathering rules and type synthesis rules to recover variable data types.

(2) Perfect and improve symbolic execution techniques so as to carry out data type recovery in intermediate language C subsets but not in compilation code. This increases system transplantability.

(3) Aimed toward improving Sub C intermediate language, ABC transformation rules are restructured. The majority of transformation rules are independent of decompilation languages and environmental objects

The authors in future will generalize 8086C language decompilation system functions, striving to resolve indeterminacy problems which exist in data type recovery, increasing man- machine mutual improvement system-user interfaces.

Acknowledgements: We are grateful for the support and guidance of Beijing Information Engineering Institute Professor Zhou Iling. In the systems in question, symbolic execution modules were implemented by Comrade Chen Kaiming, and we are grateful for this.

REFERENCES

〔1〕 Letovsky, S., A Program Anti—compiler, Proceedings of the Twenty—second Annual Hawaii International Conference on System Sciences, 1989, Vol. 2, pp505—512.

〔2〕 Duncan, R., FORTH Decompiler, Dr. Dobb's Journal (USA), 1981, Vol. 6, No. 9, pp27—41.

〔3〕 李军、张翰英，C语言反编译系统的研究与实现，计算机工程与设计，1991，No. 3，pp10—15。

〔4〕 朱三元、刘胄，国内外软件逆向工程综述，计算机应用与软件，1990，Vol. 7，No. 4，pp1—11。

〔5〕 刘宗田，68000C反编译程序的设计方法，计算机研究与发展，1986，Vol. 23，No. 6，pp32—36。

〔6〕 蔡元龙编，模式识别，西北电讯工程学院出版社，1986。

〔7〕 陈福安、刘宗田，8086C反编译系统中库函数识别技术及其实现，小型微型计算机系统，1991，Vol. 12，No. 11，pp33—40。

〔8〕 周孔伟、蔡经球，符号执行—介于程序验证和程序调试之间的方法，小型微型计算机系统，1982，No. 4。

〔9〕 刘宗田、朱逸芬，符号执行技术在68000C反编译程序中的应用，计算机学报，1988，Vol. 11，No. 10，pp633—637。

〔10〕 刘宗田、李力，8086C逆编译数据类型恢复技术，计算机研究与发展，1992，Vol. 29，No. 4，pp44—51。

〔11〕 孙永强、宋国新、杨澜，程序的结构化和变换，计算机学报，1987，Vol. 10，No. 4，pp633—637。

〔12〕 刘宗田，68000C反编译程序中的语句翻译器的设计与实现，小型微型计算机系统，1988，Vol. 9，No. 2，pp1—10。

〔13〕 刘宗田、兰群，C子集程序到C语言程序的变换，计算机研究与发展，1991，Vol. 28，No. 3，pp29—34。

DISTRIBUTION LIST

DISTRIBUTION DIRECT TO RECIPIENT

ORGANIZATION	MICROFICHE
B085 DIA/RTS-2FI	1
C509 BALLOC509 BALLISTIC RES LAB	1
C510 R&T LABS/AVEADCOM	1
C513 ARRADCOM	1
C535 AVRADCOM/TSARCOM	1
C539 TRASANA	1
Q592 FSTC	4
Q619 MSIC REDSTONE	1
Q008 NTIC	1
Q043 AFMIC-IS	1
E404 AEDC/DOF	1
E408 AFWL	1
E410 AFDTC/IN	1
E429 SD/IND	1
P005 DOE/ISA/DDI	1
P050 CIA/OCR/ADD/SD	2
1051 AFIT/LDE	1
P090 NSA/CDB	1

Microfiche Nbr: FTD95C000330L
NAIC-ID(RS)T-0024-95